3Dアクションゲームの作成

■ レイクラス

・衝突判定用のレイクラスを作成する。

Ray.h

```
#ifndef RAY_H_
#define RAY_H_

#include <gslib.h>

// レイクラス
class Ray {
public:
    // デフォルトコンストラクタ
    Ray() = default;
    // コンストラクタ
    Ray(const GSvector3& position, const GSvector3& direction) :
        position{ position }, direction{ direction } {}
    // 座標
    GSvector3 position{ 0.0f, 0.0f, 0.0f };
    // 方向
    GSvector3 direction{ 0.0f, 0.0f, 0.0f };
};

#endif
```

■ ラインクラス

・衝突判定用のラインクラスを作成する。

Line.h

```
#ifndef LINE_H_
#define LINE_H_

#include <gslib.h>

// 線分クラス
class Line {
public:
    // デフォルトコンストラクタ
    Line() = default;
    // コンストラクタ
    Line(const GSvector3& start, const GSvector3& end) :
        start{ start }, end{ end } {}
    // 始点
    GSvector3 start{ 0.0f, 0.0f, 0.0f };
    // 終点
    GSvector3 end{ 0.0f, 0.0f, 0.0f };
};

#endif
```

■ フィールドクラス

・地形・遠景を管理するフィールドクラスを作成する。

Field.h

```
#ifndef FIELD_H_
#define FIELD_H_

#include <gslib.h>

class Ray;
class Line;
class BoundingSphere;

// フィールドクラス
class Field {
public:
    // コンストラクタ
    Field(GSuint octree, GSuint skybox);
    // レイとの衝突判定
    bool collision(const Ray& ray, GSvector3* intersect = nullptr, GSplane*  plane = nullptr) const;
    // 線分との衝突判定
    bool collision(const Line& line, GSvector3* intersect = nullptr, GSplane*  plane = nullptr) const;
    // 球体との衝突判定
    bool collision(const BoundingSphere& sphere, GSvector3* intersect = nullptr) const;
    // 描画
    void draw() const;
    // コピー禁止
    Field(const Field& other) = delete;
    Field& operator = (const Field& other) = delete;
private:
    // オクツリー
    GSuint        octree_;
    // スカイボックス
    GSuint        skybox_;
};

#endif
```

Field.cpp

```
#include "Field.h"
#include "Ray.h"
#include "Line.h"
#include "BoundingSpherer.h"

// コンストラクタ
Field::Field(GSuint octree, GSuint skybox) :
    octree_{ octree }, skybox_{ skybox } {
}

// レイとの衝突判定
bool Field::collision( const Ray& ray, GSvector3* intersect, GSplane* plane) const {
    return gsOctreeCollisionRay(gsGetOctree(octree_), &ray.position, &ray.direction, intersect, plane) == GS_TRUE;
}

// 線分との衝突判定
bool Field::collision(const Line& line, GSvector3* intersect, GSplane*  plane) const {
    return gsOctreeCollisionLine(gsGetOctree(octree_), &line.start, &line.end, intersect, plane) == GS_TRUE;
}

// 球体との衝突判定
bool Field::collision(const BoundingSphere& sphere, GSvector3* intersect) const {
    return gsOctreeCollisionSphere(gsGetOctree(octree_), &sphere.center, sphere.radius, intersect) == GS_TRUE;
}

// 描画
void Field::draw() const {
    gsDrawSkyBox(skybox_);
    gsDrawOctree(octree_);
}
```

■ カメラクラス

・視点の制御を行うカメラクラスを作成する。

Camera.h

```
#ifndef CAMERA_H_
#define CAMERA_H_

#include "Actor.h"

// 一人称カメラクラス
class Camera : public Actor {
public:
    // コンストラクタ
    explicit Camera(IWorld& world);
    // 更新
    void update(float delta_time) override;
    // 描画
    void draw() const override;
};

#endif
```

Camera.cpp

```
#include "Camera.h"
#include "IWorld.h"
#include "ActorGroup.h"

// コンストラクタ
Camera::Camera(IWorld& world) {
    world_ = &world;
    name_  = "Camera";
 }

// 更新
void Camera::update(float) {
    const auto player = world_->find_actor(ActorGroup::Player, "Player");
    if (player == nullptr) return;
    position_ = player->position() + GSvector3{ 0.0f, 2.0f, 0.0f };
    rotation_ = player->rotation();
}

// 描画
void Camera::draw() const {
    glMatrixMode(GL_MODELVIEW);
    glLoadMatrixf(pose().convertViewRH());
}
```

■ プレーヤクラスの作成

・テスト用のプレーヤクラスの作成。

Player.h

```
#ifndef PLAYER_H_
#define PLAYER_H_

#include "Actor.h"

// プレーヤクラス
class Player : public Actor {
public:
    // コンストラクタ
    Player(IWorld& world, const GSvector3& position);
    // 更新
    virtual void update(float delta_time) override;
    // 描画
    virtual void draw() const override;
};

#endif
```

Player.cpp

```
#include "Player.h"
#include "IWorld.h"
#include "Field.h"
#include "Ray.h"

// コンストラクタ
Player::Player(IWorld& world, const GSvector3& position) {
    world_    = &world;
    name_     = "Player";
    position_ = position;
    body_     = BoundingSphere{ 2.0f };
}

// 更新
void Player::update(float delta_time) {
    // 移動処理
    float yaw{ 0.0f };
    if (gsGetKeyState(GKEY_LEFT) == GS_TRUE) {
        yaw = 1.0f;
    }
    if (gsGetKeyState(GKEY_RIGHT) == GS_TRUE) {
        yaw = -1.0f;
    }
    rotation_.rotate(yaw * delta_time, rotation_.getAxisY());
    float speed{ 0.0f };
    if (gsGetKeyState(GKEY_UP) == GS_TRUE) {
        speed = 1.0f;
    }
    if (gsGetKeyState(GKEY_DOWN) == GS_TRUE) {
        speed = -1.0f;
    }
    position_ += rotation_.getAxisZ() * speed * delta_time;
    // 地面との衝突判定
    Ray ray{position_, GSvector3{0.0f, 1.0f, 0.0f}};
    GSvector3 intersect;
    if (world_->field().collision(ray, &intersect)) {
        position_ = intersect;
    }
}

// 描画
void Player::draw() const {
    glPushMatrix();
        glMultMatrixf(pose());
        gsDrawMesh(0);
    glPopMatrix();
}
```

■ 敵クラスの作成

・テスト用の敵クラスの作成。

Enemy.h

```
#ifndef ENEMY_H_
#define ENEMY_H_

#include "Actor.h"

// 敵クラス
class Enemy : public Actor {
public:
    // コンストラクタ
    Enemy(IWorld& world, const GSvector3& position);
    // 更新
    virtual void update(float delta_time) override;
    //  描画
    virtual void draw() const override;
    // 衝突処理
    virtual void react(Actor& other)  override;
};

#endif
```

Enemy.cpp

```
#include "Enemy.h"
#include "IWorld.h"
#include "ActorGroup.h"
#include "EventMessage.h"
#include "Explosion.h"
#include "Field.h"
#include "Ray.h"

// コンストラクタ
Enemy::Enemy(IWorld& world, const GSvector3& position) {
    world_    = &world;
    name_     = "Enemy";
    position_ = position;
    body_     = BoundingSphere{ 2.0f };
}

// 更新
void Enemy::update(float delta_time) {
    // プレイヤーを検索
    const auto player = world_->find_actor(ActorGroup::Player, "Player");
    if (player != nullptr) {
        // プレイヤーとの距離を判定
        if (position_.distance(player->position()) < 10.0f) {
            // プレイヤーに向かって移動する
            const auto target = (player->position() - position_).getNormalized();
            position_ += target * 0.1f * delta_time;
            rotation_.identity().setAxisZ(target).nomalizeAxisZ();
        }
    }
    // 地面との衝突判定
    const Ray ray{position_, {0.0f, 1.0f, 0.0f}};
    GSvector3 intersect;
    if (world_->field().collision(ray, &intersect)) {
        position_ = intersect;
    }
}

// 描画
void Enemy::draw() const {
    glPushMatrix();
        glMultMatrixf(pose());
        gsDrawMesh(0);
    glPopMatrix();
}
```

```
// 衝突処理
void Enemy::react(Actor&) {
    // 死亡する
    die();
}
```

■ 爆発エフェクトのクラスの作成

・爆発の炎を表現するクラスを作成。

Fire.h

```
#ifndef FIRE_H_
#define FIRE_H_

#include "Actor.h"

// 炎エフェクト
class Fire : public Actor {
public:
    // コンストラクタ
    Fire(IWorld& world, const GSvector3& position);
    // 更新
    virtual void update(float delta_time) override;
    // 描画
    virtual void draw() const override;
private:
    // 消滅タイマ
    float          timer_{20.0f};
};

#endif
```

Fire.cpp

```
#include "Fire.h"

// コンストラクタ
Fire::Fire(IWorld& world, const GSvector3& position) {
    world_    = &world;
    name_     = "Fire";
    position_ = position;
    // 移動量をランダムで決定
    const auto pitch = gsRandf(0.0f, 360.0f);
    const auto yaw   = gsRandf(0.0f, 360.0f);
    const auto speed = gsRandf(0.2f, 0.5f);
    velocity_ = GSvector3::createFromPitchYaw(pitch, yaw) * speed;
}
// 更新
void Fire::update(float delta_time) {
    position_ += velocity_ * delta_time;
    velocity_ *= 0.8f * delta_time;
    if (timer_ <= 0.0f) {
        die();
    }
    timer_ -= delta_time;
}
// 描画
void Fire::draw() const {
    // 各種レンダリングモードの退避
    glPushAttrib(GL_ENABLE_BIT | GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glDisable(GL_LIGHTING);             // ライティングをオフにする
    glDepthMask(GL_FALSE);              // ｚバッファに書き込みを行わない
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);  // 加算のブレンドを有効にする
    glEnable(GL_BLEND);
    // ３Ｄスプライトの表示
    static const  GSrect body{-4.0f, 4.0f, 4.0f, -4.0f};
    const GScolor color{1.0f, 1.0f, 1.0f, timer_ / 20.0f};
    gsDrawSprite3D(0, &position_, &body, NULL, &color, NULL, 0.0f);
    // 各種レンダリングモードの復帰
    glPopAttrib();
}
```

・炎を発生させる爆発エフェクトクラスを作成する。

Explosion.h

```
#ifndef EXPLOSION_H_
#define EXPLOSION_H_

#include "Actor.h"

// 爆発エフェクト
class Explosion : public Actor {
public:
    // コンストラクタ
    Explosion(IWorld& world, const GSvector3& position);
    // 更新
    void update(float delta_time) override;
};

#endif
```

Explosion.cpp

```
#include "Explosion.h"
#include "IWorld.h"
#include "ActorGroup.h"
#include "Fire.h"

// コンストラクタ
Explosion::Explosion(IWorld& world, const GSvector3& position) :
    Actor{world, "Explosion", position, BoundingSphere{ 0.0f }} {
}

// 更新
void Explosion::update(float) {
    const int FireMax{ 10 };
    // 炎を生成する
    for (int i = 0; i < FireMax; ++i) {
        world_->add_actor(ActorGroup::Effect, new_actor<Fire>(*world_, position_));
    }
    // 生成したら死亡
    die();
}
```

■ 三人称視点カメラの作成

・三人称視点カメラを作成する。バネの計算によってカメラの座標を補間する。

TpsCamera.h

```
#ifndef TPS_CAMERA_H_
#define TPS_CAMERA_H_

#include "Actor.h"

// 三人称カメラクラス
class TpsCamera : public Actor {
public:
    // コンストラクタ
    TpsCamera(IWorld* world);
    // 更新
    virtual void update(float delta_time) override;
    // 描画
    virtual void draw() const override;
    // カメラの移動
    void move(
        const GSvector3& rest_position, // バネの静止位置
        float stiffness,                // バネ定数（バネの強さ）
        float friction,                 // 摩擦力
        float mass                      // 質量
    );
private:
    // 目標位置
    GSvector3      target_{0.0f, 0.0f, 0.0f};
};

#endif
```

TpsCamera.cpp

```
#include "TpsCamera.h"
#include "IWorld.h"
#include "ActorGroup.h"

// コンストラクタ
TpsCamera::TpsCamera(IWorld& world) {
    world_ = &world;
    name_  = "Camera";
    body_  = BoundingSphere{ 1.0f };
}

// 更新
void TpsCamera::update(float delta_time) {
    const auto player = world_->find_actor(ActorGroup::Player, "Player");
    if (player == nullptr) return;
    const auto position = GSvector3{ 0.0f, 5.0f, -15.0f } * player->pose();
    target_ = player->position() + GSvector3{ 0.0f, 4.0f, 0.0f };
    move(position, 1.0f, 0.2f, 0.8f);
}

// 描画
void TpsCamera::draw() const {
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(
        position_.x, position_.y, position_.z,
        target_.x, target_.y, target_.z,
        0.0f, 1.0f, 0.0f
    );
}
```

```
// カメラの移動
void TpsCamera::move(
    const GSvector3& rest_position,     // バネの静止位置
    float stiffness,                    // バネ定数（バネの強さ）
    float friction,                     // 摩擦力
    float mass                          // 質量
) {
    // バネの伸び具合を計算
    const auto stretch = position_ - rest_position;
    // バネの力を計算
    const auto force = -stiffness * stretch;
    // 加速度を追加
    const auto acceleration = force / mass;
    // 移動速度を計算
    velocity_ = friction * (velocity_ + acceleration);
    // 座標の更新
    position_ += velocity_;
}
```

■ ゲームプレイシーンクラス

・ゲームプレイシーンのヘッダファイル。

GamePlayScene.h

```
#ifndef GAME_PLAY_SCENE_H_
#define GAME_PLAY_SCENE_H_

#include "IScene.h"
#include "World.h"

class GamePlayScene : public IScene {
public:
    // コンストラクタ
    GamePlayScene() = default;
    // 開始
    void start() override;
    // 更新
    void update(float delta_time) override;
    // 描画
    void draw() const override;
    // 終了しているか？
    bool is_end() const  override;
    // 次のシーンを返す
    Scene next() const  override;
    // 終了
    void end() override;
private:
    World          world_;              // ワールドクラス
    bool           is_end_{ false };    // 終了フラグ
};

#endif
```

・ゲームプレイシーンのソースファイル。

GamePlayScene.cpp

```
#include "GamePlayScene.h"
#include "Scene.h"
#include "Field.h"
#include "ActorGroup.h"
#include "World.h"
#include "Player.h"
#include "Enemy.h"
#include "Camera.h"
#include "Light.h"
#include <gslib.h>

// 開始
void GamePlayScene::start() {
    is_end_ = false;

    gsLoadMesh(0, "res/dog.msh");
    gsLoadMesh(1, "res/スカイボックス.msh");
    gsLoadOctree(0, "res/地形.oct");
    gsLoadTexture(0, "res/fire.bmp");

    world_.initialize();
    world_.add_field(new_field<Field>(0, 1));
    world_.add_actor(ActorGroup::Player, new_actor<Player>(world_, GSvector3{ 0.0f, 0.0f, 0.0f }));
    world_.add_actor(ActorGroup::Enemy, new_actor<Enemy>(world_, GSvector3{ 0.0f, 0.0f, 20.0f }));
    world_.add_actor(ActorGroup::Enemy, new_actor<Enemy>(world_, GSvector3{ 20.0f, 0.0f, 0.0f }));
    world_.add_actor(ActorGroup::Enemy, new_actor<Enemy>(world_, GSvector3{ -20.0f, 0.0f, 0.0f }));
    world_.add_camera(new_actor<Camera>(world_));
    world_.add_light(new_actor<Light>(world_, GSvector3{ 1.0f, 1.0f, 1.0f }));
}

// 更新
void GamePlayScene::update(float delta_time) {
    world_.update(delta_time);
    if (gsGetKeyTrigger(GKEY_SPACE) == GS_TRUE) {
        is_end_ = true;
    }
}

// 描画
void GamePlayScene::draw() const {
    world_.draw();
}

// 終了しているか？
bool GamePlayScene::is_end() const {
    return is_end_;
}

// 次のシーンを返す
Scene GamePlayScene::next() const {
    return Scene::Title;
}

// 終了
void GamePlayScene::end() {
    world_.clear();
    gsDeleteMesh(0);
    gsDeleteMesh(1);
    gsDeleteOctree(0);
    gsDeleteTexture(0);
}
```

■ アクターマネージャの変更

・アクターを巡回するメンバ関数を追加する。

ActorManager.h

```
#ifndef ACTOR_MANAGER_H_
#define ACTOR_MANAGER_H_

#include "ActorPtr.h"
#include <list>
#include <functional>

enum class EventMessage;

// アクターマネージャー
class ActorManager {
public:
    // コンストラクタ
    ActorManager() = default;
    // アクターの追加
    void add(const ActorPtr& actor);
    // 更新
    void update(float delta_time);
    // 描画
    void draw() const;
    // 衝突判定
    void collide();
    // 衝突判定
    void collide(Actor& other);
    // 衝突判定
    void collide(ActorManager& other);
    // 削除
    void remove();
    // アクターの検索
    ActorPtr find(const std::string& name) const;
    // アクター数を返す
    unsigned int count() const;
    // アクターの巡回
    void each(std::function<void(const ActorPtr&)> fn) const;
    // メッセージ処理
    void handle_message(EventMessage message, void* param);
    // 消去
    void clear();
    // コピー禁止
    ActorManager(const ActorManager& other) = delete;
    ActorManager& operator = (const ActorManager& other) = delete;
private:
    using ActorList = std::list<ActorPtr>;
    // アクターリスト
    ActorList   actors_;
};

#endif
```

```
// アクターの巡回
void ActorManager::each(std::function<void(const ActorPtr&)> fn) const {
    for (const auto& actor : actors_) {
        fn(actor);
    }
}
```

■ アクターグループマネージャの変更

ActorGroupManager.h

```
// アクターグループマネージャ
class ActorGroupManager {
public:
    // コンストラクタ
    ActorGroupManager() = default;
    // グループの追加
    void add(ActorGroup group);
    // アクターの追加
    void add(ActorGroup group, const ActorPtr& actor);
    // 更新
    void update(float delta_time);
    // 描画
    void draw() const;
    // 描画
    void draw(ActorGroup group) const;
    // 消去
    void clear();
    // アクターを検索
    ActorPtr find(ActorGroup group, const std::string& name) const;
    // アクター数を返す
    unsigned int count(ActorGroup group) const;
    // アクターの巡回
    void each(ActorGroup group, std::function<void (const ActorPtr&)> fn) const;
    // 衝突判定
    void collide(ActorGroup group);
    // 衝突判定
    void collide(ActorGroup group1, ActorGroup group2);
    // 削除
    void remove();
    // メッセージ処理
    void handle_message(EventMessage message, void* param);
    // コピー禁止
    ActorGroupManager(const ActorGroupManager& other) = delete;
    ActorGroupManager& operator=(const ActorGroupManager& other) = delete;

private:
    using ActorGroupMap = std::map<ActorGroup, ActorManager>;
    // アクターグループマップ
    ActorGroupMap   actor_group_map_;
};
```

ActorGroupManager.cpp

```
// アクターの巡回
void ActorGroupManager::each(ActorGroup group, std::function<void(const ActorPtr&)> fn) const {
    actor_group_map_.at(group).each(fn);
}
```

■ ワールド抽象インターフェースの変更

IWorld.h

```
#ifndef IWORLD_H_
#define IWORLD_H_

#include "ActorPtr.h"
#include <string>
#include <functional>

enum class ActorGroup;
enum class EventMessage;
class Field;

// ワールド抽象インターフェース
class IWorld {
public:
    // 仮想デストラクタ
    virtual ~IWorld() {}
    // アクターを追加
    virtual void add_actor(ActorGroup group, const ActorPtr& actor) = 0;
    // アクターを検索
    virtual ActorPtr find_actor(ActorGroup group, const std::string& name) const = 0;
    // アクター数の取得
    virtual unsigned int count_actor(ActorGroup group) const = 0;
    // アクターの巡回
    virtual void each_actor(ActorGroup group, std::function<void (const ActorPtr&)> fn) const = 0;
    // メッセージを送信
    virtual void send_message(EventMessage message, void* param = nullptr) = 0;
    // フィールドを取得
    virtual Field& field() = 0;
};

#endif
```

■ ワールドクラスの変更

World.h

```
#ifndef WORLD_H_
#define WORLD_H_

#include "IWorld.h"
#include "ActorGroupManager.h"
#include "ActorPtr.h"
#include "FieldPtr.h"
#include <functional>

enum class EventMessage;

// ワールドクラス
class World : public IWorld {
public:
    // イベントメッセージリスナー
    using EventMessageListener = std::function<void(EventMessage, void*)>;
public:
    // コンストラクタ
    World();
    // 初期化
    void initialize();
    // 更新
    void update(float delta_time);
    // 描画
    void draw() const;
    // メッセージ処理
    void handle_message(EventMessage message, void* param);
    // メッセージリスナーの登録
    void add_event_message_listener(EventMessageListener listener);
    // フィールドの追加
    void add_field(const FieldPtr& field);
```

```
    // カメラの追加
    void add_camera(const ActorPtr& camera);
    // ライトの追加
    void add_light(const ActorPtr& light);
    // 消去
    void clear();

    // アクターの追加
    virtual void add_actor(ActorGroup group, const ActorPtr& actor) override;
    // アクターの検索
    virtual ActorPtr find_actor(ActorGroup group, const std::string& name) const override;
    // アクター数の取得
    virtual unsigned int count_actor(ActorGroup group) const override;
    // アクターの巡回
    virtual void each_actor(ActorGroup group, std::function<void(const ActorPtr&)> fn) const override;
    // メッセージの送信
    virtual void send_message(EventMessage message, void* param = nullptr) override;
    // フィールドを取得
    virtual Field& field() override;

    // コピー禁止
    World(const World& other) = delete;
    World& operator = (const World& other) = delete;
private:
    // アクターマネージャー
    ActorGroupManager   actors_;
    // ライト
    ActorPtr            light_;
    // カメラ
    ActorPtr            camera_;
    // フィールド
    FieldPtr            field_;
    // メッセージリスナ
    EventMessageListener listener_ { [](EventMessage, void*) {} };
};

#endif
```

World.cpp

```
// アクターの巡回
void World::each_actor(ActorGroup group, std::function<void(const ActorPtr&)> fn) const {
    actors_.each(group, fn);
}
```

■ 3Dアクションゲームクラス一覧

衝突判定

BoundingSpherer.h	BoundingSphere.cpp	衝突判定用の球体
Line.h	Line.cpp	衝突判定用の線分
Ray.h	Ray.cpp	衝突判定用のレイ

アクター・ワールド

Actor.h	Actor.cpp	全キャラクタの基底クラス
ActorPtr.h		アクタークラス用シェアドポインタ
ActorManager.h	ActorManager.cpp	複数のアクターを管理する
ActorGroupManager.h	ActorGroupManager.cpp	アクターのグループを管理する
ActorGroup.h		アクターの種類を表す列挙型
IWorld.h		アクター用のワールドインターフェース
World.h	World.cpp	ゲームの世界を管理する
EventMessage.h		イベントメッセージの種類を表す列挙型

フィールド・ライト・カメラ

Field.h	Field.cpp	地形・スカイボックスの管理
FieldPtr.h		フィールドクラス用シェアドポインタ
Light.h	Light.cpp	3D 空間上のライト
Camera.h	Camera.cpp	3D 空間上の 1 人称カメラ
TpsCamera.h	TpsCamera.cpp	3D 空間上の３人称カメラ

キャラクタ

Player.h	Player.cpp	プレーヤキャラクタ
Enemy.h	Enemy.cpp	敵キャラクタ
Explosion.h	Explosion.cpp	爆発キャラクタ
Fire.h	Fire.cpp	爆発炎キャラクタ

シーン管理

SceneManager.h	SceneManager.cpp	シーンの管理をする
IScene.h		シーン抽象インターフェース
IScenePtr.h		シーン用のシェアドポインタクラス
SceneNull.h	SceneNull.cpp	ヌルシーンクラス（ダミーシーン）
Scene.h		シーン名を表す列挙型

タイトルシーン・ゲーム中シーン

TitleScene.h	TitleScene.cpp	タイトルシーン
GamePlayScene.h	GamePlayScene.cpp	ゲームプレイ中シーン

アプリケーション

Game.h	Game.cpp	ゲームアプリケーションクラス

main 関数

	main.cpp	main 関数用